Bounded Size Dictionary Compression: Relaxing the LRU Deletion Heuristic

Sergio De Agostino

Computer Science Department
Università "La Sapienza"
Via Salaria 113, 00135 Roma, Italy

e-mail: deagostino@di.uniroma1.it

Abstract. The unbounded version of the Lempel-Ziv dynamic dictionary compression method is P-complete. Therefore, it is unlikely to implement it with sublinear work space unless a deletion heuristic is applied to bound the dictionary. The well-known LRU strategy provides the best compression performance among the existent deletion heuristics. We show experimental results on the compression effectiveness of a relaxed version (RLRU) of the LRU heuristic. RLRU partitions the dictionary in p equivalence classes, so that all the elements in each class are considered to have the same "age" for the LRU strategy. Such heuristic turns out to be as good as LRU when p is greater or equal to 2. Moreover, RLRU is slightly easier to implement than LRU in addition to be more space efficient.

Keywords: Bounded dictionary compression, space complexity, LRU strategy

1 Introduction

The Lempel-Ziv dynamic dictionary (LZ2) compression algorithm learns substrings by reading the input string from left to right with an *incremental parsing* procedure [7]. The dictionary is empty, initially. The procedure adds a new substring to the dictionary as soon as a prefix of the still unparsed part of the string does not match a dictionary element and replaces the prefix with a pair comprising a pointer to the dictionary and the last uncompressed character. For example, the parsing of the string $abababaaaaaa$ is a, b, ab, aba, aa, aaa and the coding is $0a, 0b, 1b, 3a, 1a, 5a$ (the pointer value for the first element in the dictionary is 1 and 0 represents the empty string). We will see in the next section different LZ2 compression heuristics (NC, FC, ID, AP), which work with a dictionary containing initially the alphabet characters and produce a coding with no raw characters.

The main issue for implementation purposes is to bound the work space to produce the incremental parsing of the string to compress. Since the problem of computing such parsing is P-complete [2, 3], it is unlikely to have sublinear work space when LZ2 compression is implemented unless a deletion heuristic is applied to bound the dictionary. Several deletion heuristics have been designed and applied to the compression heuristics mentioned above (see the books of Storer [5, 6] and Bell, Cleary

and Witten [1]). A strategy that can achieve good compression ratio with small memory is the LRU deletion heuristic that discards the least recently used dictionary element to make space for the new substring. The least recently used strategy provides the best compression performance among the well-known heuristics (FREEZE, RESTART, SWAP, LRU). AP-LRU turns out to be the best compression heuristic when the dictionary is bounded.

When the size of the dictionary is $O(\log^k n)$ the LRU strategy is log-space hard for SC^k (Steve Cook's class), the class of problems solvable simultaneously in polynomial time and $O(\log^k n)$ space [4]. Since its sequential complexity is polynomial in time and $O(\log^k n \log\log n)$ in space, the problem belongs to SC^{k+1}. Moreover, in [4] a relaxed version (RLRU) was introduced which turned out to be the first (and only so far) natural SC^k-complete problem. RLRU partitions the dictionary in p equivalence classes, so that all the elements in each class are considered to have the same "age" for the LRU strategy.

While in [4] the RLRU heuristic was considered only for theoretical reasons concerning complexity theory, in this paper we want to look at its practical aspects. We show experimental results on its compression effectiveness for $2 \leq p \leq 6$, using the AP compression heuristic. RLRU turns out to be as good as LRU even when p is equal to 2. Since RLRU removes an arbitrary element from the equivalence class with the "older" elements, the two classes (when p is equal to 2) can be implemented with a couple of stacks, which makes RLRU slightly easier to implement than LRU in addition to be more space efficient. Surprisingly, the compression effectiveness (which we can measure as the inverse of the compression ratio) is not monotonicly increasing with the value of p. This might be explained by the fact that the approach is heuristic (choosing to remove an older element is not always a better choice). However, LRU is always strictly better (in an irrelevant way for the compression effectiveness) than RLRU. This fact shows that there should be always an improvement when two values of p differ substantially.

Simpler choices for the deletion heuristic are FREEZE, RESTART and SWAP. These heuristics do not delete elements from the dictionary at each step. SWAP is the best among these simpler approaches and has a worse compression performance than RLRU and LRU. We describe compression and deletion heuristics in section 2. In section 3, we discuss the complexity of the LRU and RLRU heuristics. In section 4, we compare the experimental results of LRU, RLRU and SWAP. Conclusions are given in section 5.

2 Compression and Deletion Heuristics

As mentioned in the introduction, the compression and deletion heuristics presented in this section can be found in [1, 5, 6]. The incremental parsing procedure used by the LZ2 algorithm produces a compressed string comprising pointers and raw characters. In practice, we do not want to leave characters uncompressed. This can be avoided by initializing the dictionary with the alphabet characters. The NC (next character) heuristic also parses the string from left to right with a greedy procedure. It finds the longest match in the current position and updates the dictionary by adding the concatenation of the match with the next character. The FC (first character) heuristic differs in the way it updates the dictionary. The element to add is defined

as the concatenation of the last match with the first character of the current match. With the ID (identity) heuristic, the element to add is defined as the concatenation of the last match with the whole current match. The AP (all prefixes) heuristic adds a set of elements to the dictionary at each step. Each element is the concatenation of the last match with a prefix of the current match. In this way, the dictionary of the AP heuristic has both the characteristics of the dictionaries of the FC and ID heuristics. Observe that with FC, ID and AP, an element to add might be in the dictionary already. How these heuristics work on the example in the introduction is shown in Figure 1.

NC heuristic

parsing: a, b, ab, aba, a, aa, aa;

dictionary: $a, b, ab, ba, aba, abaa, aa, aaa$;

coding: 1, 2, 3, 5, 1, 7, 7

FC heuristic

parsing: a, b, ab, ab, a, a, aa, aa

dictionary: a, b, ab, ba, aba, aa, aaa, aaaa

coding: 1, 2, 3, 3, 1, 1, 7, 7

ID heuristic

parsing: a, b, ab, ab, a, a, aa, aa

dictionary : a, b, ab, bab, abab, aba, aa, aaaa

coding: 1, 2, 3, 3, 1, 1, 7, 7

AP heuristic

parsing: a, b, ab, ab, a, a, aa, aa

dictionary : a, b, ab, ba, bab, aba, abab, aa, aaa, aaaa

coding: 1, 2, 3, 3, 1, 1, 8, 8

Figure 1: The compression heuristics.

It is well known that these heuristics can be implemented by storing the dictionary in a tree data structure, called *trie*. At each step, we find the longest match in the dictionary as a path from the root to a leaf of the trie and update the dictionary by adding a new leaf to the trie. Real time implementations are possible for each compression heuristic using any deletion heuristic (FREEZE, RESTART, SWAP, LRU and RLRU) to bound the dictionary. FREEZE, RESTART and SWAP work as it follows:

- FREEZE: once the dictionary is full, freeze it and do not allow any further entries to be added.

- RESTART: stop adding further entries when the dictionary is full; when the compression ratio starts deteriorating clear the dictionary and learn new strings.
- SWAP: when the *primary* dictionary first becomes full, start an *auxiliary* dictionary, but continue compression based on the primary dictionary; when the auxiliary dictionary becomes full, clear the primary dictionary and reverse their roles.

The SWAP and RESTART heuristics can be viewed as discrete versions of LRU. In fact, the dictionaries depend only on small segments of the input string.

parsing: $a, b, ab, ab, a, a, aa, aa$;

dictionary (step 3): a, b, ab, ba, bab

dictionary (step 4): a, b, ab, ba, aba

dictionary (step 4): a, b, ab, abab , aba

dictionary (step 6): a, b, ab, aa, aba

dictionary (step 7): a, b, ab, aa, aaa

dictionary (step 8): a, b, aaaa, aa, aaa

coding: 1, 2, 3, 3, 1, 1, 4, 4

Figure 2: The AP-LRU heuristic on the example string.

We showed in the introduction of the paper how the LZ2 algorithm parses the example string *abababaaaaaa*. If we bound the dictionary size with 3 and use LRU, after three steps a, b, ab is the partial parsing, $0a, 0b, 1b$ is the partial coding and the dictionary is filled up with the three elements a, b, ab. The LRU heuristic works as follows:

> LRU: define a string as "used" when it is added to the dictionary and remove the least recently used leaf of the trie representing the dictionary after a new leaf is added. The pointer to the element which is removed becomes the pointer to the new element.

Hence at the fourth step, first *aba* is added and coded as $3a$. Then, b is discarded. Finally, *aba* is replaced with *aa*, coded as $1a$, and *ab* with *aaa*, coded as 2a.

Observe that while for the NC heuristic the element added to the dictionary is an extension of the current match as for the original LZ2 algorithm, this is not true for the other heuristics. To make things work properly when we apply the LRU deletion strategy to the FC, ID and AP heuristics, a string is defined to be "used" also when it is matched. AP-LRU turns out to be the best compression heuristic when the dictionary is bounded. How the AP-LRU heuristic works on the example string with a dictionary of size 5 is shown in Figure 2. Steps correspond to the parsing. In this example, the AP-LRU heuristic adds more than one element only at the fourth parsing step. In Figure 3, we extend the example by adding the suffix *bbaaa* to make some observations. At step 11, the current match is removed from the dictionary. In

this case, the AP-LRU heuristic puts it back into the dictionary at step 12 and then it adds its extensions (this can happen with FC and ID as well). With AP, it could be possible that prefixes of the current match are removed and similarly they would be put back into the dictionary at the next step. Finally, observe that at step 10 if *aab* were removed instead of *aaa*, *aaa* would be parsed off at the end providing a shorter code for the string. This shows that removing the older element might not be the better choice.

parsing: $a, b, ab, ab, a, a, aa, aa, b, b, aa, a$;

dictionary (step 9): a, b, aab, aa, aaa

dictionary (step 10): a, b, aab, aa, bb

dictionary (step 11): a, b, ba, aa, bb

dictionary (step 11): a, b, ba, baa, bb

dictionary (step 12): a, b, ba, baa, aa

dictionary (step 12): a, b, ba, aa, aaa

coding: 1, 2, 3, 3, 1, 1, 4, 4, 2, 2, 4, 1

Figure 3: The AP-LRU heuristic on the extended example.

We present, now, a relaxed version of LRU. The relaxed version (RLRU) of the LRU heuristic is:

> RLRU: When the dictionary is not full, label the i^{th} element added to the dictionary with the integer $\lceil i \cdot p/k \rceil$, where k is the dictionary size minus the alphabet size and $p < k$ is the number of labels. When the dictionary is full, label the $i-th$ element with p if $\lceil i \cdot p/k \rceil = \lceil (i-1)p/k \rceil$. If $\lceil i \cdot p/k \rceil > \lceil (i-1)p/k \rceil$, decrease by 1 all the labels greater or equal to 2. Then, label the $i-th$ element with p. Finally, remove one of the elements represented by a leaf with the smallest label.

In other words, RLRU works with a partition of the dictionary in p classes, sorted somehow in a fashion according to the order of insertion of the elements in the dictionary, and an arbitrary element from the oldest class with removable elements is deleted when a new element is added. RLRU is more sophisticated than SWAP (which is the best among the simpler deletion strategies presented above) since it removes elements in a continuous way as the original LRU. In fact, we will see in section 4 that the compression performance of AP-RLRU is better than AP-SWAP. Moreover, even if it relaxes on the choice of the element to remove AP-RLRU is as good as AP-LRU.

3 The Complexity of LRU and RLRU Heuristics

The unbounded version of the LZ2 compression method is P-complete [2, 3]. This means there is a log-space reduction from any problem in P to the problem of computing LZ2 compression. Since it is believed that POLYLOGSPACE, the class of problems computed with polylogarithmic work space, is not contained in P, it is unlikely to have sublinear work space when LZ2 compression is implemented unless a deletion heuristic is applied to bound the dictionary.

The LZ2 algorithm with LRU deletion heuristic on a dictionary of size $O(\log^k n)$ can be performed in polynomial time and $O(\log^k n \log\log n)$ space (n is the length of the input string). In fact, the trie requires $O(\log^k n)$ space by using an array implementation since the number of children for each node is bounded by the alphabet cardinality. The $\log\log n$ factor is required to store the information needed for the LRU deletion heuristic since each node must have a different age, which is an integer value between 0 and the dictionary size. Obviously, this is true for any LZ2 heuristic (NC, FC, ID, AP). If the size of the dictionary is $O(\log^k n)$, the LRU strategy is log-space hard for SC^k (Steve Cook's class), the class of problems solvable simultaneously in polynomial time and $O(\log^k n)$ space [4]. The problem belongs to SC^{k+1}. This hardness result is not so relevant for the space complexity analysis since $\Omega(\log^k n)$ is an obvious lower bound to the work space needed for the computation. Much more interesting is what can be said about the parallel complexity analysis. In [4] it was shown that LZ2 compression using the LRU deletion heuristic with a dictionary of size c can be performed in parallel either in $O(\log n)$ time with $2^{O(c\log c)}n$ processors or in $2^{O(c\log c)}\log n$ time with $O(n)$ processors. This means that if the dictionary size is constant, the compression problem belongs to NC, the class of problems solvable in polylogarithmic time with a polynomial number of processors. NC and SC (the class of problems solvable simultaneously in polynomial time with polylogarithmic work space) are classes that can be viewed in some sense symmetric and are believed to be incomparable. Since log-space reductions are in NC, the compression problem cannot belong to NC when the dictionary size is polylogarithmic if NC and SC are incomparable. We want to point out that the dictionary size c figures as an exponent in the parallel complexity of the problem. This is not by accident. If we believe that SC is not included in NC, then the SC^k-hardness of the problem when c is $O(\log^k n)$ implies the exponentiation of some increasing and diverging function of c. In fact, without such exponentiation either in the number of processors or in the parallel running time, the problem would be SC^k-hard and in NC when c is $O(\log^k n)$. Observe that the P-completeness of the problem, which requires a superpolylogarithmic value for c, does not suffice to infer this exponentiation since c can figure as a multiplicative factor of the time function. Moreover, this is a unique case where somehow we use hardness results to argue that practical algorithms of a certain kind (NC in this case) do not exist because of huge multiplicative constant factors occurring in their analysis.

Finally, the LZ2 compression heuristics with RLRU deletion heuristic on a dictionary of size $O(\log^k n)$ can be performed in polynomial time and $O(\log^k n)$ space since the number of ages is constant. In fact, LZ2-RLRU compression is the first (and only so far) natural SC^k-complete problem [4].

4 Experimental Results

We show experimental results concerning the compression effectiveness of AP-RLRU with a number of classes between 2 and 6, and compare them with the results of AP-SWAP and AP-LRU. Each class is implemented with a stack. Therefore, the newest element in the class of least recently used elements is removed. Observe that if RLRU worked with only one class, after the dictionary is filled up the next element added would be immediately deleted. Therefore, RLRU would work like FREEZE. This is why we show results for a number of classes between 2 and 6. We considered natural language, programming language and postscript. The dictionary size in real life implementations has usually varied between 4,096 (twelve bits pointer size) and 65,536 (sixteen bits pointer size). In Figure 4, we present results with a dictionary size equal to 4,096.

Heuristic	English	C Programs	Postscript
LRU	.51034	.52026	.46806
RLRU2	.51193	.52039	.46971
RLRU3	.51147	.52060	.46916
RLRU4	.51153	.51957	.46902
RLRU5	.51159	.52008	.46888
RLRU6	.51150	.51982	.46919
SWAP	.68654	.71967	.61341

Figure 4: Compression ratios with dictionary size 4,096.

We experimented on samples of English text files, C programs and Postscript files. The file size varied between 100 Kilobytes and 2 Megabytes. The table shows the average of the compression ratios obtained on each sample. RLRUp denotes that the RLRU heuristic works with p classes. The compression ratios of LRU and RLRUp for $2 \leq p \leq 6$ are about the same up to the third or fourth decimal digit. On the other hand, their compression effectiveness provides about 15 to 20 percent improvement on the performance of SWAP. As mentioned in the introduction, the compression effectiveness of the RLRU heuristic is not monotonicly increasing with the value of p, which might be explained by the fact that the approach is heuristic (choosing to remove an older element is not always a better choice as discussed with the example of Figure 3).

The compression ratios of LRU and RLRU improve when the dictionary size is 65,536 as shown in Figure 5, but they compare to each other in a similar way while SWAP is only a 3 percent of LRU and RLRU on C programs and about 10 and 20 percent on English and Postscript, respectively.

5 Conclusions

We showed that a relaxed version of the best bounded size dictionary LZ2 compression technique, which uses the least recently used strategy, provides the same compression effectiveness. This version is more space efficient and easier to implement, since it

Heuristic	English	C Programs	Postscript
LRU	.32363	.38213	.33556
RLRU2	.32371	.38221	.33710
RLRU3	.32414	.38219	.33667
RLRU4	.32374	.38229	.33613
RLRU5	.32342	.38217	.33588
RLRU6	.32349	.38216	.33597
SWAP	.41402	.41657	.52827

Figure 5: Compression ratios with dictionary size 65,536.

relaxes by making a bipartition of the dictionary which defines, generally speaking, a set of less recently used elements from which one element can be removed arbitrarily.

References

[1] Bell, T.C., J.G. Cleary and I.H. Witten [1990]. *Text Compression*, Prentice Hall.

[2] De Agostino, S. [1994]. "P-complete Problems in Data Compression", *Theoretical Computer Science* **127**, 181-186.

[3] De Agostino, S. [2000]. "Erratum to P-complete Problems in Data Compression", *Theoretical Computer Science* **234**, 325-326.

[4] De Agostino, S. and R. Silvestri [2003]. "Bounded Size Dictionary Compression: SC^k-Completeness and NC Algorithms", *Information and Computation* **180**, 101-112.

[5] Storer, J.A. [1988]. *Data Compression: Methods and Theory* (Computer Science Press).

[6] Storer, J.A. [1992]. "Massively Parallel Systolic Algorithms for Real-Time Dictionary-Based Text Compression" *Image and Text Compression*, Kluwer Accademic Publishers (Storer J.A., editor), 159–178.

[7] Ziv, J. and A. Lempel [1978]. "Compression of Individual Sequences via Variable Rate Coding", *IEEE Transactions on Information Theory* **24**, 530-536.